Low-rank passthrough neural networks

Low-rank passthrough neural networks

Antonio Valerio Miceli Barone *
School of Informatics, The University of Edinburgh
amiceli@inf.ed.ac.uk

Abstract

Various common deep learning architectures, such as LSTMs, GRUs, Resnets and Highway Networks, employ state passthrough connections that support training with high feed-forward depth or recurrence over many time steps. These "Passthrough Networks" architectures also enable the decoupling of the network state size from the number of parameters of the network, a possibility has been studied by Sak et al. (2014) with their low-rank parametrization of the LSTM. In this work we extend this line of research, proposing effective, low-rank and low-rank plus diagonal matrix parametrizations for Passthrough Networks which exploit this decoupling property, reducing the data complexity and memory requirements of the network while preserving its memory capacity. This is particularly beneficial in low-resource settings as it supports expressive models with a compact parametrization less susceptible to overfitting. We present competitive experimental results on several tasks, including language modeling and a near state of the art result on sequential randomly-permuted MNIST classification, a hard task on natural data.

1 Overview

Deep neural networks can perform non-trivial computations by the repeated the application of parametric non-linear transformation layers to vectorial data. This staging of many computation steps can be done over a time dimension for tasks involving sequential inputs or outputs of varying length, yielding a *recurrent neural network*, or over an intrinsic circuit depth dimension, yielding a *deep feed-forward neural network*, or both. Training these deep models is complicated by the *exploding* and *vanishing* gradient problems (Hochreiter, 1991; Bengio et al., 1994).

Various network architectures have been proposed to ameliorate the vanishing gradient problem in the recurrent setting, such as the LSTM (Hochreiter and Schmidhuber, 1997; Graves and Schmidhuber, 2005), the GRU (Cho et al., 2014), the RHN (Zilly et al., 2016), etc. Similar methods have also been applied in the feed-forward setting with architectures such as Highway Networks (Srivastava et al., 2015), Deep Residual Networks (He et al., 2015), and so on. All these architectures are based on a single structural principle which, in this work, we will refer to as the *state passthrough*. We will thus refer to these architectures as *Passthrough Networks*.

In many settings, especially low-resource natural language processing tasks, the main difficulty in training neural networks is the trade-off between the network representation power and its training data complexity, which is related to the number of trainable parameters.

On one hand, the number of parameters can be thought as the number of tunable "knobs" that need to be set to represent a function, on the other hand, it also constrains the size of the partial results that are propagated inside the network. In typical fully connected networks, a layer acting on a n-dimensional state vector has $O(n^2)$ parameters stored in one or more matrices, but there can be many functions of practical interest that are simple enough to be represented by a relatively small number of bits while still requiring some sizable amount of memory in order to be computed. Therefore, representing these functions on

* Work partially done while affiliated with University of Pisa.

Proceedings of the Workshop on Deep Learning Approaches for Low-Resource NLP, pages 77–86
Melbourne, Australia July 19, 2018.

a fully connected neural network can be wasteful in terms of number of parameters. The full parameterization implies that, at each step, all the information in each state component can affect all the information in any state component at the next step. Classical physical systems, however, consist of spatially separated parts with primarily local interactions, long-distance interactions are possible but they tend to be limited by propagation delays, bandwidth and noise. Therefore it may be beneficial to bias our model class towards models that tend to adhere to these physical constraints by using a parametrization which reduces the number of parameters required to represent them. This can be accomplished by imposing some constraints on the $n \times n$ matrices that parametrize the state transitions. One way of doing this is to impose a convolutional structure on these matrices (LeCun et al., 2004; Krizhevsky et al., 2012), which corresponds to strict locality and periodicity constraints as in a cellular automaton. These constraints work well in certain domains such as vision, but may be overly restrictive in other domains.

The state passthrough allows for a systematic decoupling of the network state size from the number of parameters: since by default the state vector passes mostly unaltered through the layers, each layer can be made simple enough to be described only by a small number of parameters without affecting the overall memory capacity of the network, effectively spreading the computation over the depth or time dimension of the network, but without making the network "thin". This has been exploited by some convolutional passthrough architectures (Srivastava et al., 2015; He et al., 2015; Kaiser and Sutskever, 2015), or architectures with addressable read-write memory (Graves et al., 2014; Danihelka et al., 2016).

In this work we propose simple but effective low-dimensional parametrizations that exploit this decoupling based on low-rank or low-rank plus diagonal matrix decompositions. Our approach extends the LSTM architecture with a single projection layer proposed by Sak et al. (2014) which has been applied to speech recognition, natural language modeling (Józefowicz et al., 2016), video analysis (Sun et al., 2015), etc. We provide experimental evaluation of our approach on GRU and LSTM architectures on various machine learning tasks, including a near state of the art result for the hard task of sequential randomly-permuted MNIST image recognition (Le et al., 2015).

2 Model

2.1 Passthrough networks

A (fixed-width) neural network can be described as a dynamical system with a n-dimensional state vector $x(t) \in \mathcal{R}^n$ that transforms an input u into an output y over multiple time steps T.

Passthrough networks can be defined as networks where the state evolves according to a transition function f which has a special form such that, at each step t the state vector $x(t)$ is propagated to the next step modified only by some (nearly) linear, element-wise transformation.

We define a network to have a *state passthrough* on x if x evolves as

$$x(t) = \pi(t) \odot \tau(t) + x(t-1) \odot \gamma(t) \qquad (1)$$

where π is the *next state proposal*, τ is the *state transform*, γ is the *state carry* and $\odot$ denotes element-wise vector multiplication.

Additional non-passthrough state vectors may be also present[1].

As concrete example, we can describe the fully connected Gated Recurrent Unit (GRU) by Cho et al. (2014) as

$$\begin{aligned}
\omega(t) &= \sigma(\theta^{W_\omega} u(t) + \theta^{U_\omega} x(t-1)) \\
\gamma(t) &= \sigma(\theta^{W_\gamma} u(t) + \theta^{U_\gamma} x(t-1)) \\
\tau(t) &= 1^{\otimes n} - \gamma(t) \\
\pi(t) &= g(\theta^{W_\pi} u(t) + \theta^{(U_\pi)}(x(t-1) \odot \omega(t)))
\end{aligned} \qquad (2)$$

where g is the hyperbolic tangent, σ is the logistic sigmoid, $\theta^{W_\omega}, \theta^{W_\gamma}, \theta^{W_\pi} \in \mathcal{R}^{n \times m}$ are input parameter matrices and $\theta^{U_\omega}, \theta^{U_\gamma}, \theta^{U_\pi} \in \mathcal{R}^{n \times n}$ are the recurrent parameter matrices (bias vectors θ^b– not shown). $\omega(t)$ is the *reset gate* which is specific to the GRU architecture.

2.2 Low-rank passthrough networks

We can impose a low-rank constraint on the state transition matrices by rewriting each of them as the product of two matrices, where the inner dimension d is a model hyperparameter.

In the case of the GRU of eq. 2 we can redefine

[1] For instance the LSTM has a passthrough "cell" state and a non-passthrough "hidden" state.

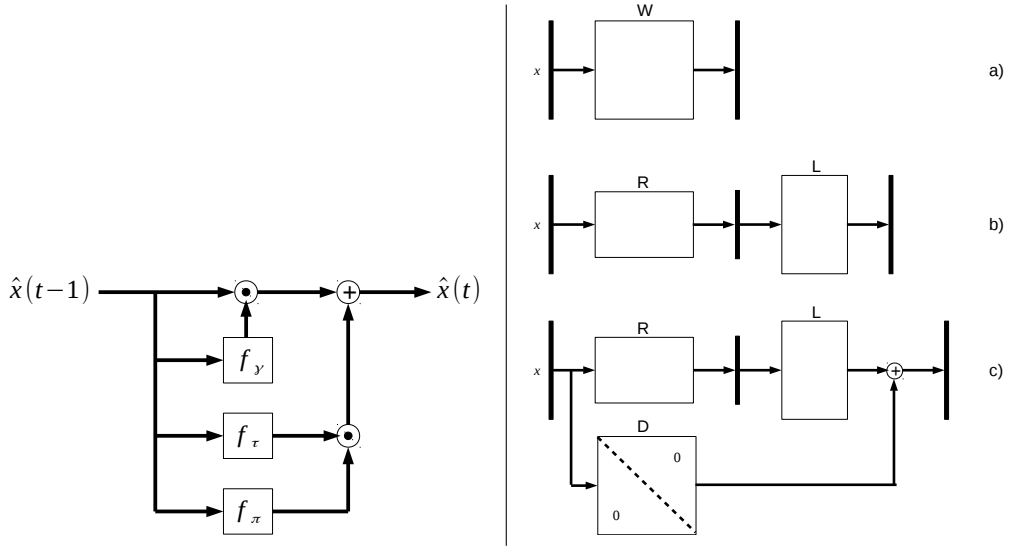

Figure 1: Left: Generic state passthrough hidden layer, optional per-timestep input $u(t)$ is not shown. Right: a) Full matrix parametrization. b) Low-rank parametrization. c) Low-rank plus diagonal parametrization.

the recurrent parameter matrices as

$$\begin{aligned}\theta^{(U_\omega)} &= \theta^{(L_\omega)} \cdot \theta^{(R_\omega)} \\ \theta^{(U_\gamma)} &= \theta^{(L_\gamma)} \cdot \theta^{(R_\gamma)} \\ \theta^{(U_\pi)} &= \theta^{(L_\pi)} \cdot \theta^{(R_\pi)}\end{aligned} \tag{3}$$

where $\theta^{(L_\_)} \in \mathcal{R}^{n \times d}$ and $\theta^{(R_\_)}, \in \mathcal{R}^{d \times n}$. When $d < n/2$ this result in a reduction of the number of trainable parameters of the model.

Even when $n/2 \leq d < n$, while the total number of parameter increases, the number of degrees of freedom of the model still decreases, because low-rank factorization are unique only up to arbitrary $d \times d$ invertible matrices, thus the number of independent degrees of freedom of a low-rank layer is $2nd - d^2$.

This low-rank constraint can be thought as a bandwidth constraint on the computation performed at each step: the R matrices first project the state into a smaller subspace, extracting the information needed for that specific step, then the L matrices project it back to the original state space, spreading the selected information to all the state components that need to be updated.

In this parametrization, which we denote as *untied low-rank*, we allow each parameter matrix to be parametrized independently by a pair of R and L matrices. This extends the approach of Sak et al. (2014) for the LSTM architecture, which we denote here as *tied low-rank*, where they instead force the R matrices to be the same for all the functions of the state transition.

A low-rank parametrization can be also applied to the input matrices of recurrent neural networks, in fact, the input embedding layer commonly used in NLP applications results in a tied low-rank parametrization of the input whenever the embedding size is lower than the RNN state size.

Low-rank passthrough architectures are universal in that they retain the same representation classes of their parent architectures. This equivalence can be realized in the worst case by exploiting a depth-rank tradeoff (e.g. either $O(n)$ rank and $O(T)$ depth or $O(1)$ rank and $O(nT)$ depth)[2].

2.3 Low-rank plus diagonal passthrough networks

As we show in the experimental section, on some tasks the low-rank constraint may prove to be excessively restrictive if the goal is to train a model with fewer parameters than one with arbitrary matrices. A simple extension is to add to each low-rank parameter matrix a diagonal parameter matrix, yielding a matrix that is full-rank but still parametrized in a low-dimensional space. For instance, for the GRU we modify eq. 3 to

$$\begin{aligned}\theta^{(U_\omega)} &= \theta^{(L_\omega)} \cdot \theta^{(R_\omega)} + \theta^{(D_\omega)} \\ \theta^{(U_\gamma)} &= \theta^{(L_\gamma)} \cdot \theta^{(R_\gamma)} + \theta^{(D_\gamma)} \\ \theta^{(U_\pi)} &= \theta^{(L_\pi)} \cdot \theta^{(R_\pi)} + \theta^{(D_\pi)}\end{aligned} \tag{4}$$

[2]In the case of recurrent networks, depth is intended as the *recurrence depth* (Zilly et al., 2016).

where $\theta^{(D_-)}$ are trainable diagonal parameter matrices.

It may seem that adding diagonal parameter matrices is redundant in passthrough networks. After all, the state passthrough itself can be considered as a diagonal matrix applied to the state vector, which is then additively combined to the new proposed state computed by the f_π function. However, since the state passthrough completely skips over all non-linear activation functions, these formulations are not equivalent. In particular, the low-rank plus diagonal parametrization may help in recurrent neural networks which receive input at each time step, since it allows each component of the state vector to directly control how much input signal is inserted into it at each step. We demonstrate the effectiveness of this model for the sequence copy, sequential MNIST and language modeling tasks described in the experiments section.

3 Experiments

We applied the Low-rank GRU (LR-GRU) and Low-rank plus diagonal GRU (LRD-GRU) architectures to a subset of sequential benchmarks described in the Unitary Evolution Recurrent Neural Networks (uRNN) article by Arjovsky et al. (2015), specifically the memory task, the addition task and the sequential randomly permuted MNIST task. For the memory tasks, we also considered two different variants proposed by Danihelka et al. (2016) and Henaff et al. (2016) which are hard for the uRNN architecture. We chose to compare against the uRNN architecture because it set state of the art results in terms of both data complexity and accuracy and because it is an architecture with similar design objectives as low-rank passthrough architectures, namely a low-dimensional parametrization and the mitigation of the vanishing gradient problem, but it is based on quite different principles.

We also applied these architectures to a character-level language modeling task on the Penn Treebank corpus. For the language modeling task, we also experimented with Low-rank plus diagonal LSTMs.

3.1 Memory task

The input of an instance of this task is a sequence of $T = N + 20$ discrete symbols in a ten symbol alphabet $a_i : i \in 0, \ldots 9$, encoded as one-hot vectors. The first 10 symbols in the sequence are "data" symbols i.i.d. sampled from $a_0, \ldots, a_7$, followed by $N-1$ "blank" a_8 symbols, then a distinguished "run" symbol a_9, followed by 10 more "blank" a_8 symbols. The desired output sequence consists of $N + 10$ "blank" a_8 symbols followed by the 10 "data" symbols as they appeared in the input sequence. Therefore the model has to remember the 10 "data" symbol string over the temporal gap of size N, which is challenging for a recurrent neural network when N is large. In our experiment we set $N = 500$, which is the hardest setting explored in the uRNN work. The training set consists of $100,000$ training examples and $10,000$ validation/test examples. The architecture is a GRU with a dense $n \times 10$ output matrix followed a (biased) softmax. We train to minimize the cross-entropy loss.

We were able to solve this task using a GRU with full recurrent matrices with state size $n = 128$, learning rate 1×10^{-3}, mini-batch size 20, initial bias of the carry functions (the "update" gates) 4.0, however this model has many more parameters, nearly $50,000$ in the recurrent layer only, than the uRNN work which has about $6,500$, and it converges much more slowly than the uRNN. We were not able to achieve convergence with a pure low-rank model without exceeding the number of parameters of the fully connected model, but we achieved fast convergence with a LRD-GRU model with $d = 50$, with other hyperparameters set as above. This model has still more parameters ($39,168$ in the recurrent layer, $41,738$ total) than the uRNN model and converges more slowly but still reasonably fast, reaching test cross-entropy $< 1 \times 10^{-3}$ nats and almost perfect classification accuracy in less than $35,000$ updates.

In order to obtain a fair comparison, we also train a uRNN model with state size $n = 721$, resulting in approximately the same number of parameters as the LRD-GRU models. This model very quickly reaches perfect accuracy on the training set in less than $2,000$ updates, but overfits w.r.t. the test set.

We also consider two variants of this task which are difficult for the uRNN model. For both these tasks we used the same settings as above except that the task size parameter is set at $N = 100$ for consistency with the works that introduced these variants. In the variant of Danihelka et al. (2016), the length of the sequence to be remembered is

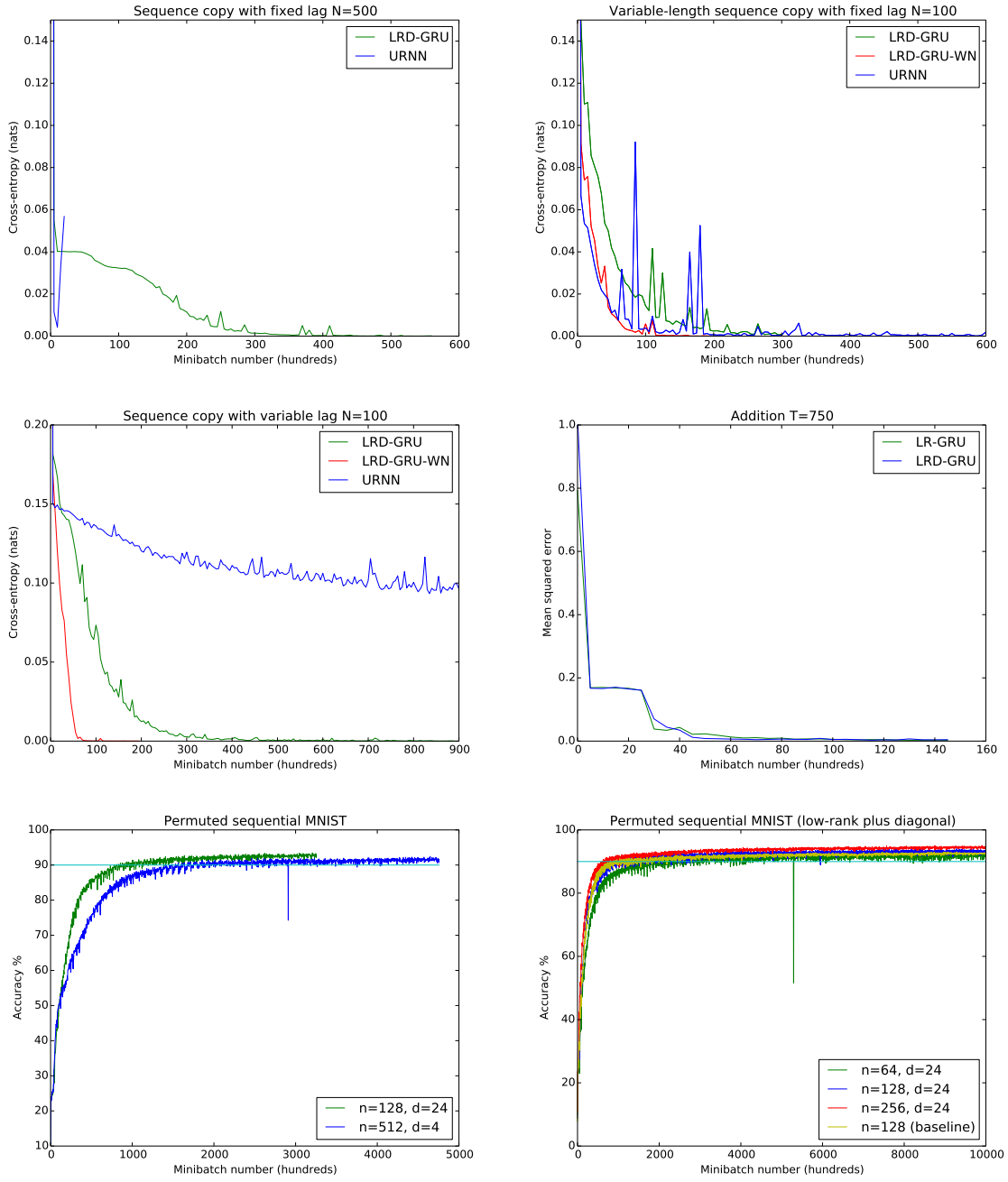

Figure 2: Top row and middle left: LRD-GRU and uRNN on the sequence copy tasks, cross-entropy on validation set. Middle right: LR-GRU and LRD-GRU on the addition task, mean squared error on validation set. Bottom row: LR-GRU (left) and LRD-GRU (right) on the permuted sequential MNIST task, accuracy on validation set, horizontal line indicates 90% accuracy.

randomly sampled between 1 and 10 for each sequence. They manage to achieve fast convergence with their Associative LSTM architecture with $65,505$ parameters, and slower convergence with standard LSTM models. Our LRD-GRU architecture, which has less parameters than their Associative LSTM, performs comparably or better, reaching test cross-entropy $< 1 \times 10^{-3}$ nats and almost perfect classification accuracy in less than $30,000$ updates. In the variant of Henaff et al. (2016), the length of the sequence to be remembered is fixed at 10 but the model is expected to copy it after a variable number of time steps randomly chosen, for each sequence, between 1 and $N = 100$. The authors achieve slow convergence with a standard LSTM model, while our LRD-GRU architecture achieves fast convergence, reaching test cross-entropy $< 1 \times 10^{-3}$ nats and almost perfect classification accuracy in less than $38,000$ updates, and perfect test accuracy in $87,000$ updates.

We further train uRNN models with state size $n = 721$ on these variants of the memory task. We found that the uRNN learns faster than the LRD-GRU on the variable length, fixed lag task (Danihelka et al., 2016) but fails to converge within our training time limit on the fixed length, variable lag task (Henaff et al., 2016).

Training the LRD-GRU on these tasks incurs sometimes in numerical stability problems as discussed in sec. 3.5.1. In order to systemically address these issues, we also trained models with weight normalization (Salimans and Kingma, 2016) and weight row max-norm constraints. These models turned out to be more stable and in fact converge faster, performing on par with the uRNN on the variable length, fixed lag task.

Training curves are shown in figure 2 (top and middle left).

3.2 Addition task

For each instance of this task, the input sequence has length T and consists of two real-valued components, at each step the first component is independently sampled from the interval $[0, 1]$ with uniform probability, the second component is equal to zero everywhere except at two randomly chosen time step, one in each half of the sequence, where it is equal to one. The result is a single real value computed from the final state which we want to be equal to the sum of the two elements of the first component of the sequence at the positions where the second component was set at one. In our experiment we set $T = 750$.

The training set consists of $100,000$ training examples and $10,000$ validation/test examples. We use a LR-GRU and a LRD-GRU with $2 \times n$ input matrix, $n \times 1$ output matrix and (biased) identity output activation. We train to minimize the mean squared error loss. We use state size $n = 128$, maximum rank $d = 24$. This results in approximately $6,140$ parameters in the recurrent hidden layer. Learning rate was set at 1×10^{-3}, mini-batch size 20, initial bias of the carry functions (the "update" gates) was set to 4.

We trained on $14,500$ mini-batches, obtaining a mean squared error on the test set of 0.003 for both parametrizations, which is a better result than the one reported in the uRNN article, in terms of training time and final accuracy. The training curves are shown in figure 2 (middle right).

3.3 Sequential MNIST task

This task consists of handwritten digit classification on the MNIST dataset with the caveat that the input is presented to the model one pixel value at time, over $T = 784$ time steps. To further increase the difficulty of the task, the inputs are reordered according to a random permutation (fixed for all the task instances).

We use LR-GRUs and a LRD-GRUs with $1 \times n$ input matrix, $n \times 10$ output matrix and (biased) softmax output activation. Learning rate was set at 5×10^{-4}, mini-batch size 20, initial bias of the carry functions (the "update" gates) was set to 5.

Results are presented in table 1 and training curves are shown in figure 2 (bottom row). All these models except the one with the most extreme bottleneck ($n = 512, d = 4$) exceed the reported uRNN test accuracy of 91.4%, although they converge more slowly (hundred of thousands updates vs. tens of thousands of the uRNN). Also note that the LRD-GRU is more accurate than the full-rank GRU with the same state size, while the LR-GRU is slightly less accurate (in terms of test accuracy), indicating the utility of the diagonal component of the parametrization for this task.

These are on par with more complex architectures with time-skip connections (Zhang et al., 2016) (reported test set accuracy 94.0%). To our knowledge, at the time of this writing, the best result on this task is the LSTM with recurrent batch normalization by Cooijmans et al. (2016)

Table 1: Sequential permuted MNIST results

Architecture	state size	max rank	params	val. accuracy	test accuracy
Baseline GRU	128	-	51.0 k	93.0%	92.8%
LR-GRU	128	24	20.2 k	93.4%	91.8%
LR-GRU	512	4	19.5 k	92.5%	91.3%
LRD-GRU	64	24	10.3 k	93.1%	91.9%
LRD-GRU	128	24	20.6 k	94.1%	93.5%
LRD-GRU	256	24	41.2 k	**95.1%**	**94.7%**

(reported test set accuracy 95.2%). The architectural innovations of these works are orthogonal to our own and in principle they can be combined to it.

3.4 Character-level language modeling task

This standard benchmark task consist of predicting the probability of the next character in a sentence after having observed the previous charters. Following Zaremba et al. (2014), we use the Penn Treebank English corpus, with standard training, validation and test splits. As a baseline we use a single layer GRU either with no regularization or regularized with Bayesian recurrent dropout (Gal, 2015).

In our experiments we primarily consider the LRD-GRU, both with tied and untied projection matrices. We set the state size and maximum rank to either reduce the total number of parameters compared to the baselines or to keep the number of parameters approximately the same while increasing the memory capacity. We also compare with the LR-GRU. Results are shown in table 2.

Our LRD-GRU reduces the model per-character perplexity (the base-2 exponential of the bits-per-character entropy). Both the tied and untied versions perform equally when the state size is the same, but the tied version performs better when the number of parameters is kept the same, presumably due to the increased memory capacity of the state vector. Our best model has an extreme bottleneck, over a hundred of times smaller than the state size, while the word-level language models trained by Józefowicz et al. (2016) use bottlenecks of four to eight times smaller than the state size. This difference is likely due to our usage of the "plus diagonal" parametrization, in fact, the plain LR-GRU with such an extreme bottleneck fails to even approach the baselines.

In terms of absolute perplexity, our results are worse than published ones (e.g. Graves (2013)), although they may not be directly comparable since published results generally use different training and evaluation schemes, such as preserving the network state between different sentences.

We ran additional experiments using LSTM architectures, similar to Graves (2013), although we still could not obtain the same baseline performance even using the Adam optimizer (using SGD+momentum yields even worse results). In fact, we obtained approximately the same perplexity as our baseline GRU model with the same state size.

We applied the Low-rank plus diagonal parametrizations to our LSTM architecture maintaining the same number of parameters as the baseline. We obtained notable perplexity improvements over the baseline.

3.5 Experimental details

3.5.1 Low-rank GRUs

In our experiments (except language modeling) we optimized using RMSProp (Tieleman and Hinton, 2012) with gradient component clipping at 1. Code is available online [3]. Our code is based on the published uRNN code[4] (specifically, on the LSTM implementation) by the original authors for the sake of a fair comparison. In order to achieve convergence on the memory task however, we had to slightly modify the optimization procedure, specifically we changed gradient component clipping with gradient norm clipping (with NaN detection and recovery), and we added a small $\epsilon = 1 \times 10^{-8}$ term in the parameter update formula. No modifications of the original optimizer implementation were required for the other tasks.

In order to address the numerical instability issues in the memory tasks, we also consider a variant of our LRD-GRU where apply weight nor-

[3] https://github.com/Avmb/lowrank-gru

[4] https://github.com/amarshah/complex_RNN

Table 2: Character-level language modeling results

Architecture	dropout	tied	state size	max rank	params	test char perplexity
Baseline GRU	No	-	1000	-	3.11 M	2.96
Baseline GRU	Yes	-	1000	-	3.11 M	2.92
Baseline GRU	Yes	-	3298	-	33.0 M	2.77
Baseline LSTM	Yes	-	1000	-	4.25 M	2.92
LRD-GRU	No	No	1000	64	0.49 M	2.92
LRD-GRU	No	No	3298	128	2.89 M	2.95
LRD-GRU	Yes	No	3298	128	2.89 M	2.86
LRD-GRU	Yes	No	5459	64	2.69 M	2.82
LRD-GRU	Yes	Yes	5459	64	1.99 M	2.81
LRD-GRU	No	Yes	1000	64	0.46 M	2.90
LRD-GRU	Yes	Yes	4480	128	2.78 M	2.86
LRD-GRU	Yes	Yes	6985	64	2.54 M	**2.76**
LR-GRU	Yes	Yes	6985	64	2.54 M	9.88
LRD-LSTM	Yes	No	1740	300	4.25 M	2.86

malization as described by Salimans and Kingma (2016) to all the parameter matrices except the output one and the diagonal matrices. All these matrices have trainable scale parameters, except for the projection (R) matrices. We further apply a hard constraint on the matrices row norms by clipping them at 10 after each update. We disable NaN detection and recovery during training. The rationale behind this approach, in addition to the general benefits of normalization, is that the low-rank parametrization potentially introduces stability issues because the model is invariant to the simultaneous multiplication of a row of an R-matrix by a scalar s and the division of the corresponding column of the L-matrix by s, which in principle allows the parameters of either matrix to grow very large in magnitude, eventually resulting in overflows or other pathological behavior. The weight row max-norm constraint can counter this problem. But the constraint alone could make the optimization problem harder by restricting and distorting the parameter space. Fortunately we can counter this by weight normalization which makes the model invariant to the row-norms of the parameter matrices.

In the language modeling experiment, the character vocabulary size is 51, we use no character embeddings. Training is performed with Adam with learning rate 1×10^{-3}. Bayesian recurrent dropout was adapted from the original LSTM architecture of Gal (2015) to the GRU architecture as in Sennrich et al. (2016).

Our implementation is based on the "dl4mt" tutorial[5] and the Nematus neural machine translation system [6]. The code is available online [7].

3.5.2 Low-rank LSTMs

For our LSTM experiments, we modified the implementation of LSTM language model with Bayesian recurrent dropout by Gal (2015)[8]. In order to match the setup of Graves (2013) more closely, we used a vocabulary size of 49, no embedding layer and one LSTM layer. We the Adam optimizer with learning rate 2×10^{-4}. The low-rank plus diagonal parametrization is applied on the recurrence matrices as in the GRU models. The code is available online[9].

4 Conclusions

We proposed low-dimensional parametrizations for passthrough neural networks based on low-rank or low-rank plus diagonal decompositions of the $n \times n$ matrices that occur in the hidden layers. We experimentally compared our models with state of the art models, obtaining competitive results including a near state of the art for the randomly-permuted sequential MNIST task,

[5] https://github.com/nyu-dl/dl4mt-tutorial

[6] https://github.com/EdinburghNLP/nematus

[7] https://github.com/Avmb/dl4mt-lm/tree/master/lm

[8] https://github.com/yaringal/BayesianRNN

[9] https://github.com/Avmb/lowrank-lstm

and improvements over the baselines on a language modeling task. We showed that the LRD parametrization outperforms the LR parametrization in almost all task and never underpeforms it, which highlight as the main contribution of this work. Therefore recommend to always include a diagonal parameter matrix whenever a low-rank parametrization is used. We also presented a weight row-norm constraint trick to improve optimization stability for these kind of architectures with multiplicative symmetries.

References

Martin Arjovsky, Amar Shah, and Yoshua Bengio. 2015. Unitary evolution recurrent neural networks. *CoRR*, abs/1511.06464.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville. 2016. Recurrent Batch Normalization. *ArXiv e-prints*.

I. Danihelka, G. Wayne, B. Uria, N. Kalchbrenner, and A. Graves. 2016. Associative Long Short-Term Memory. *ArXiv e-prints*.

Yarin Gal. 2015. A theoretically grounded application of dropout in recurrent neural networks. *arXiv preprint arXiv:1512.05287*.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610.

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*.

M. Henaff, A. Szlam, and Y. LeCun. 2016. Orthogonal RNNs and Long-Memory Tasks. *ArXiv e-prints*.

Sepp Hochreiter. 1991. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*.

Lukasz Kaiser and Ilya Sutskever. 2015. Neural gpus learn algorithms. *CoRR*, abs/1511.08228.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. 2015. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.

Yann LeCun, Fu Jie Huang, and Leon Bottou. 2004. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–97. IEEE.

Hasim Sak, Andrew W Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342.

Tim Salimans and Diederik P Kingma. 2016. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *arXiv preprint arXiv:1602.07868*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Highway networks. *arXiv preprint arXiv:1505.00387*.

Chen Sun, Sanketh Shetty, Rahul Sukthankar, and Ram Nevatia. 2015. Temporal localization of fine-grained actions in videos by domain transfer from web images. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference*, pages 371–380. ACM.

Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5 - rmsprop,.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan Salakhutdinov, and Yoshua Bengio. 2016. Architectural complexity measures of recurrent neural networks. *arXiv preprint arXiv:1602.08210*.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. 2016. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*.